Building Java Programs

Chapter 8: Classes

Lecture 8-2: Constructors, Encapsulation, Critters

reading: 8.4 - 8.6

Lecture outline

- anatomy of a class, continued
  - initializing objects: constructors
  - encapsulation
    - private fields
  - printing objects: the `toString` method

Object initialization: constructors

reading: 8.4

self-check: #10-12
exercises: #9, 11, 14, 16

Initializing objects

- Currently it is tedious to create a `Point` and initialize it:

```
Point p = new Point();
p.x = 3;
p.y = 8;                          // tedious
```

- We'd rather pass the fields' initial values as parameters:

```
Point p = new Point(3, 8);    // better!
```

  - We are able to this with Java's built-in `Point` class.

Constructors

- **constructor**: Initializes the state of new objects.

```
public <type> ( <parameter(s)> ) {
    <statement(s)> ;
}
```

  - runs only when the client uses the `new` keyword
  - does not specify a return type;
    it implicitly returns the new object being created
  - If a class has no constructor, Java gives it a *default constructor* with no parameters that sets all fields to 0.

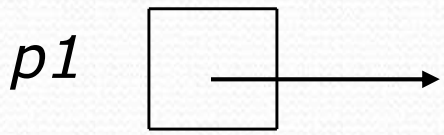
Constructor example

```
public class Point {
    int x;
    int y;

    // Constructs a Point at the given x/y location.
    public Point(int initialX, int initialY) {
        x = initialX;
        y = initialY;
    }

    public void translate(int dx, int dy) {
        x = x + dx;
        y = y + dy;
    }
}
```

Tracing a constructor call

- What happens when the following call is made?

```
Point p1 = new Point(7, 2);
```




```
 x                    y

public Point(int initialX, int initialY) {
    x = initialX;
    y = initialY;
}

public void translate(int dx, int dy) {
    x = x + dx;
    y = y + dy;
}
```

Client code, version 3

```
public class PointMain3 {
    public static void main(String[] args) {
        // create two Point objects
        Point p1 = new Point(5, 2);
        Point p2 = new Point(4, 3);

        // print each point
        System.out.println("p1: (" + p1.x + ", " + p1.y + ")");
        System.out.println("p2: (" + p2.x + ", " + p2.y + ")");

        // move p2 and then print it again
        p2.translate(2, 4);
        System.out.println("p2: (" + p2.x + ", " + p2.y + ")");
    }
}
```

OUTPUT:

```
p1: (5, 2)
p2: (4, 3)
p2: (6, 7)
```

The `toString` method

reading: 8.6

self-check: #18, 20-21

exercises: #9, 14

Printing objects

- By default, Java doesn't know how to print objects:

```
Point p = new Point(10, 7);
System.out.println("p: " + p);  // p is Point@9e8c34
```

- We can print a better string (but this is cumbersome):

```
System.out.println("p: (" + p.x + ", " + p.y + ")");
```

- We'd like to be able to print the object itself:

```
// desired behavior
System.out.println("p: " + p);  // p is (10, 7)
```

The `toString` method

- tells Java how to convert an object into a `String`
- called when an object is printed/concatenated to a `String`:

```
Point p1 = new Point(7, 2);
System.out.println("p1 is " + p1);
```

  - If you prefer, you can write `.toString()` explicitly.

```
System.out.println("p1 is " + p1.toString());
```

- Every class has a `toString`, even if it isn't in your code.
  - The default is the class's name and a hex (base-16) number:

```
Point@9e8c34
```

toString syntax

```
public String toString() {
    <code that returns a String> ;
}
```

- Example:

```
// Returns a String representing this Point.
public String toString() {
    return "(" + x + ", " + y + ")";
}
```

- The method name, return, parameters must match exactly.

- Modify our client code to use `toString`.

Client code

```
// This client program uses the Point class.
public class PointMain {
    public static void main(String[] args) {
        // create two Point objects
        Point p1 = new Point(7, 2);
        Point p2 = new Point(4, 3);

        // print each point
        System.out.println("p1: " + p1);
        System.out.println("p2: " + p2);

        // compute/print each point's distance from the origin
        System.out.println("p1's distance from origin: " + p1.distanceFromOrigin());
        System.out.println("p2's distance from origin: " + p1.distanceFromOrigin());

        // move p1 and p2 and print them again
        p1.translate(11, 6);
        p2.translate(1, 7);
        System.out.println("p1: " + p1);
        System.out.println("p2: " + p2);

        // compute/print distance from p1 to p2
        System.out.println("distance from p1 to p2: " + p1.distance(p2));
    }
}
```

Encapsulation

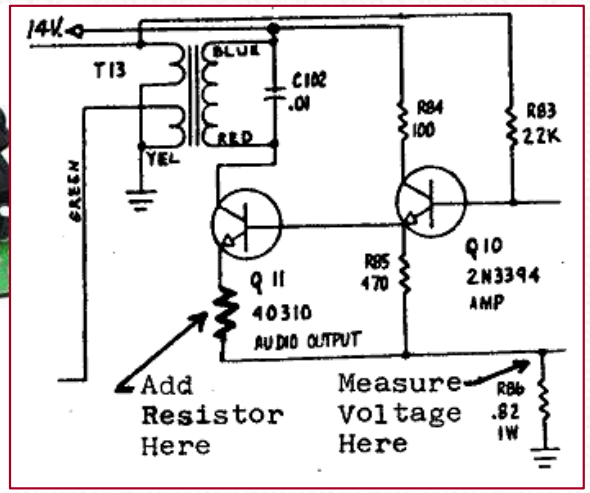
reading: 8.5 - 8.6

self-check: #13-17

exercises: #5

Encapsulation

- **encapsulation**: Hiding implementation details of an object from its clients.
  - Encapsulation provides *abstraction*.
    - separates external view (behavior) from internal view (state)



Private fields

- Fields can be declared *private*.
  - No code outside their own class can access or change them.

```
private <type> <name> ;
```

  - Examples:

```
private int x;
private String name;
```

- Client code sees an error when accessing private fields:

```
PointMain.java:11: x has private access in Point
System.out.println("p1 is (" + p1.x + ", " + p1.y + ")");
                                  ^
```

Accessing private state

- We can provide methods to get and/or set a field's value:

```
// A "read-only" access to the x field ("accessor")
public int getX() {
    return x;
}

// Allows clients to change the x field ("mutator")
public void setX(int newX) {
    x = newX;
}
```

  - Client code will look more like this:

```
System.out.println("p1: (" + p1.getX() + ", " + p1.getY() + ")");
p1.setX(14);
```

Point class, version 4

```
// A Point object represents an (x, y) location.
public class Point {
    private int x;
    private int y;

    public Point(int initialX, int initialY) {
        x = initialX;
        y = initialY;
    }

    public double distanceFromOrigin() {
        return Math.sqrt(x * x + y * y);
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    public void setLocation(int newX, int newY) {
        x = newX;
        y = newY;
    }

    public void translate(int dx, int dy) {
        x = x + dx;
        y = y + dy;
    }
}
```

Client code, version 4

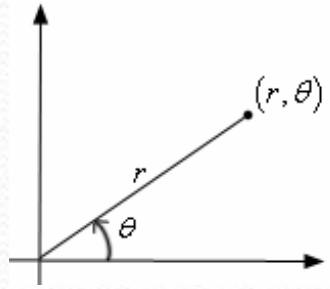
```
public class PointMain4 {
    public static void main(String[] args) {
        // create two Point objects
        Point p1 = new Point(5, 2);
        Point p2 = new Point(4, 3);

        // print each point
        System.out.println("p1: (" + p1.getX() + ", " + p1.getY() + ")");
        System.out.println("p2: (" + p2.getX() + ", " + p2.getY() + ")");

        // move p2 and then print it again
        p2.translate(2, 4);
        System.out.println("p2: (" + p2.getX() + ", " + p2.getY() + ")");
  }
}
```

OUTPUT:

```
p1 is (5, 2)
p2 is (4, 3)
p2 is (6, 7)
```

Benefits of encapsulation

- Provides abstraction between an object and its clients.
- Protects an object from unwanted access by clients.
  - A bank app forbids a client to change an `Account`'s balance.
- Allows you to change the class implementation.
  - `Point` could be rewritten to use polar coordinates (radius *r*, angle *θ*), but with the same methods.

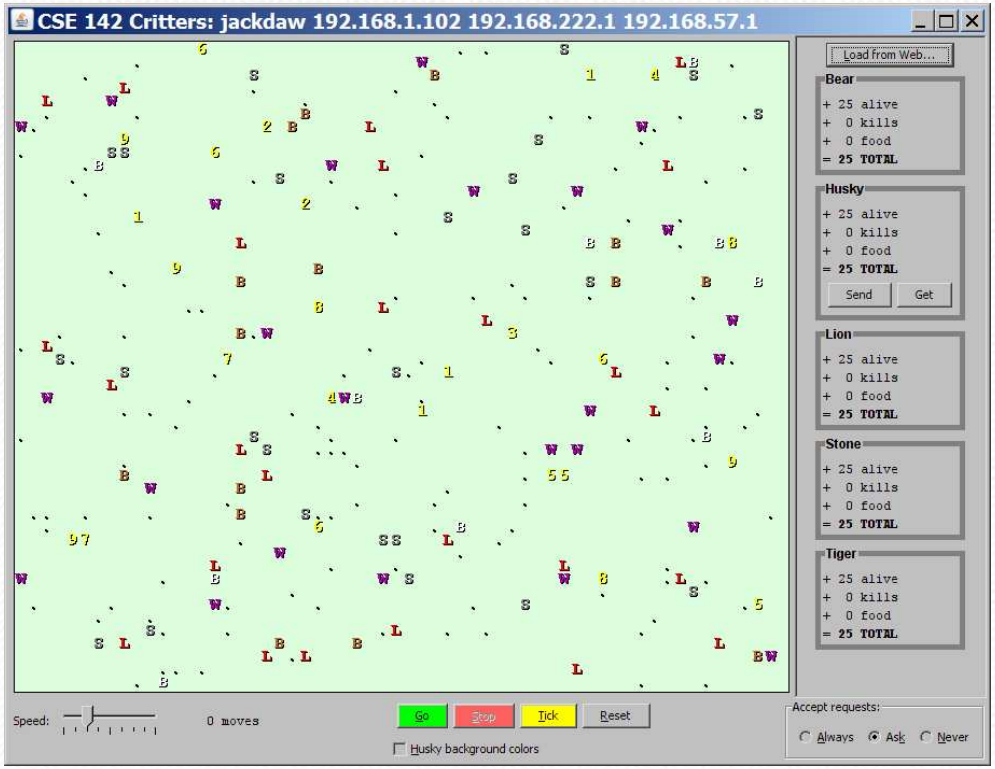


- Allows you to constrain objects' state (**invariants**).
  - Example: Only allow `Points` with non-negative coordinates.

Homework 8: Critters

Critters

- A simulation world with animal objects with behavior:
  - `getMove` movement
  - `eat` eating food
  - `fight` animal fighting
  - `toString` letter to display
  - `getColor` color to display



- You must implement:
  - `Bear`
  - `Lion`
  - `Tiger`
  - `Husky`

A Critter class

```
public class <name> extends Critter {
    ...
}
```

- Writing `extends Critter` tells the simulator that your class is a critter animal
  - This is an example of *inheritance*, which we'll see in Ch. 9

- Write some/all 5 methods to give your animals behavior.

How the simulator works

- When you press "Go", the simulator enters a loop:
  - move each animal once (`getMove`), in random order
  - if the animal has moved onto an occupied square, `fight!`
  - if the animal has moved onto food, ask it if it wants to `eat`

- Key concept: The simulator is in control, NOT your animal.
  - Example: `getMove` can return only one move at a time. `getMove` can't use loops to return a sequence of moves.

  - Your animal must keep state (as fields) so that it can make a single move, and know what moves to make later.

Critter exercise

- Write a critter class `Cougar` (the dumbest of all animals):
  - `eat`: Always eats.
  - `fight`: Always pounces.
  - `getColor`: Blue if the `Cougar` has never fought; red if he has.
  - `getMove`: The drunk `Cougar` staggers left 2, right 2, repeats.
  - `toString`: Always returns `"C"`.

Ideas for state

- Counting is often helpful:
  - How many total moves has this animal made?
  - How many times has it eaten? Fought?
- Remembering recent actions in fields is helpful:
  - Which direction did the animal move last?
    - How many times has it moved that way?
  - Did the animal eat the last time it was asked?
  - How many steps has the animal taken since last eating?
  - How many fights has the animal been in since last eating?
- You must not only have the right state, but update that state properly when relevant actions occur.

Keeping state

- How can a critter move left 2, right 2, and repeat?

```
public Direction getMove() {
    for (int i = 1; i <= 2; i++) {
        return Direction.LEFT;
    }
    for (int i = 1; i <= 2; i++) {
        return Direction.RIGHT;
    }
}
```

```
private int moves;   // total moves made by this Critter

public Direction getMove() {
    moves++;
    if (moves % 4 == 1 || moves % 4 == 2) {
        return Direction.LEFT;
    } else {
        return Direction.RIGHT;
    }
}
```

Critter solution

```
public class Cougar extends Critter {
    private int moves;
    private boolean fought;

    public Cougar() {
        moves = 0;
        fought = false;
    }

    public boolean eat() {
        return true;
    }

    public Attack fight() {
        fought = true;
        return Attack.POUNCE;
    }

    public Color getColor() {
        if (fought) {
            return Color.RED;
        } else {
            return Color.BLUE;
        }
    }

    public Direction getMove() {
        moves++;
        if (moves % 4 == 1 || moves % 4 == 2) {
            return Direction.WEST;
        } else {
            return Direction.EAST;
        }
    }

    public String toString() {
        return "C";
    }
}
```